27 12 2023
Wednesday

MAP

Map is a data structure which stores data in the form of <key, value> pairs in order sorted by key.

Implementation- Balanced BST.

Time complexity for insertion, deletion and searching is O(logn).

Unordered map

Unordered map is same as the map. The difference is it doesn't stores data in sorted order.

Implementation- Array / Hash Table / Bucket Array

Time complexity for insertion, deletion and searching is O(1).

* **Header files -**

For map = #include <map>
For unordered map = #include <unordered_map>

* Creation -

Declaration- unordered_map<string, int> mapping;

Initialisation- unordered_map<string, int> mapping{
{"one", 1}, {"two", 2}, {"three", 3}
};

Same with map.

* Insertion -

Insertion is done using pair. Pair can be created using below ways -

- pair<string, int> p = make_pair("one", 1);
- pair<string, int> q ("two", 2);
- pair<string, int> r;
  r.first = "three";
  r.second = 3;

And these pairs can be inserted in map by -

mapping_name.insert(pair_name);

= There is one more way of insertion -
map_name["four"] = 4;

* **Size -**

```
map_name.size();
```

* **Accessing values at key -**

- `map_name.at(key_name);`
- `map_name[key_name];`

* **Searching -**

- `map_name.count(key);`

This returns 1 if key is present and it returns 0 if not present.

- `map_name.find(key);`

This function returns an iterator which refers to the position in map where key is present. If no matching key present, it returns an iterator which refers to `map_name.end();`

* **Check empty -**

```
map_name.empty();
```

* **To remove an entry -**

```
map_name.erase(key);
```

* To remove all entries-

map_name.clear();

Note- If coe will search for the entry or the key for which there is no entry in the map, then it will return 0 as the value for that key and also a new entry will be created in the map for that key.

* HASH FUNCTIONS-

A hash function is a function that converts a given numeric or alphanumeric key to a small practical integer value.

Hash function comprises - • Hash code
• Compression function

Hash code - Converts input into some integer value.

Compression function - Ye us particular integer value ko inside bounds of array leke aata hai using some techniques.

Overall, the motive is input ko integer value me convert krke us value ko array ke kisi index pr store krana.

Note - There is no hash function with zero collision. There is always some collision. There are some techniques or strategies used to prevent collision.

* **Collision -**

There is possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision.

* **Collision handling techniques -**

- Open Hashing / Closed Addressing / separate chaining
- Close Hashing / open Addressing.

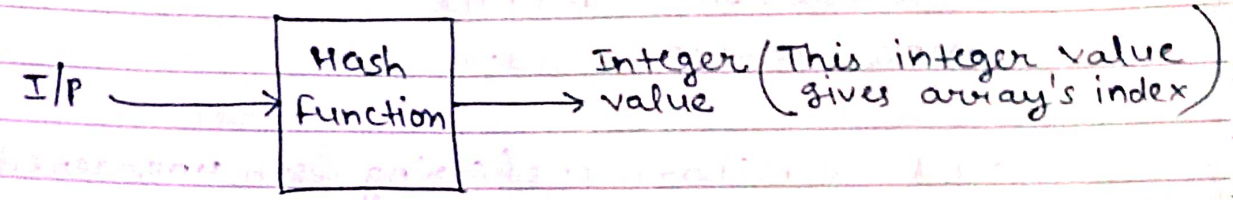
- **Open Hashing -** Basically, a linked list data structure is used to implement this technique. When a number of input keys refers to the same value or index, then they are inserted into singly linked list.

- **Close Hashing -** It stores all elements directly in the hash table. This method uses probing techniques like linear, quadratic.

Linear Probing - Searching the hash table consecutively until an empty index is found.

Quadratic probing - In which the interval between probes increases quadratically. Like if hash value is some H.f. Quadratic probing will be like $H.f + 0^2$, $H.f + 1^2$, $H.f + 2^2$, $H.f + 3^2$ --- until empty slot isn't found.

* How unordered map does searching in O(1) -



Now, length of array in which data is to be stored is n and length of input is some K. Assuming, $n >>>>> K$. So, this will take $O(K) \simeq O(1)$ time.

Now, if we know the index of the array where data is to be stored, we can insert, delete and search in O(1) time. For searching, we just need to do or find arr[index] which will take constant time.

* Load factor - $\frac{\text{no. of elements}}{\text{no. of free boxes}}$

If this ratio < 0.7, it means hash function is good.

How to make a good hash function -

- Either increase the no. of free boxes.
- Or decrease the no. of elements.

* Count appearance of each characters in string-

```
#include <bits/stdc++.h>
using namespace std;

void countCharacters(string &str, unordered_map<
                                    char, int> &mp){
    for(auto i: str){
        mp[i]++;
    }
}

int main(){
    string str= "lovebabbar";
    unordered_map<char, int> mp;

    countCharacters(str, mp);

    for(auto i:mp){
        cout<< i.first <<"-> "<<i.second << endl;
    }
}
```

o/p -

```
r -> 1
b -> 3
a -> 2
l -> 1
v -> 1
o -> 1
e -> 1
```

02 01 2024
Tuesday

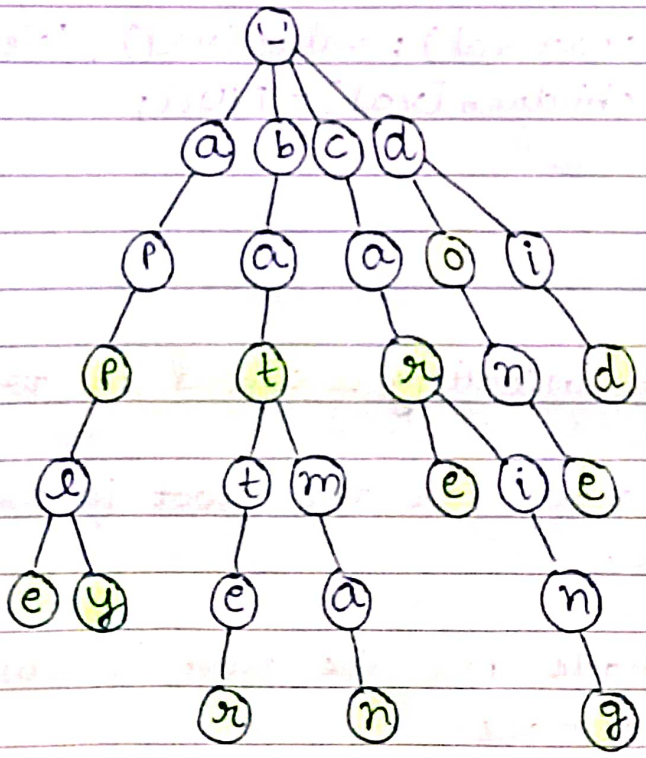
TRIE DATA STRUCTURE

Trie is a multi-way tree structure generally used for pattern searching. It is also known as prefix tree.

Suppose, we have some strings like -

app, apple, apply, bat, batter, batman, car, care, cast, do, did, done

and we have to insert these strings in tries. The trie will look like as follows -



The root node contains any special character. Every node has any number of children. The highlighted nodes represents string termination.

* A TrieNode comprises -

- Characters
- Children (can be stored using array/maps)
- isTerminal → 1/0 (It tells if the character is the last character of the string. Or we can say, string termination)

```
class TrieNode{
public:
char value;
unordered_map<char, TrieNode*> children;
bool isTerminal;

TrieNode(char val): value(val), isTerminal(0){
        children[val] = NULL;
    }
};
```

* Logic for inserting a word in trie -

- If child node for the root is present, then -
  → traverse.

- If child node for the root is absent, then -
  → create the node.
  → traverse.

* Logic for searching a word in trie -

- If the child node for the root is present. Then -
→ traverse

- If the child node for the root is absent, it means the string given for search is not present. Then -
→ return false.

* Logic for deleting a word from trie -

- If the child node for the root is present. Then -
→ traverse

- If absent, then -
→ return.

* Time complexity for insertion, deletion and searching is O(string length) which we can assume to be constant. So, we can say that T.C. is $O(k) \simeq O(1)$.

Code -

```
class TrieNode{
public:
char value;
unordered_map<char, TrieNode*> children;
bool isTerminal;

TrieNode(char val): value(val), isTerminal(0){
        children[val] = NULL;
  }
};
// For inserting coord in trie
void insertWord(TrieNode* root, string coord){
    //base case
    if(coord.length()==0){
        root->isTerminal = true;
        return;
    }
    //1case
    char ch= coord[0];
    TrieNode* child=NULL;

    if(root->children[ch]){
        //present
        child = root->children[ch];
    }
    else{
        //absent
        child = new TrieNode(ch);
        root->children[ch] = child;
    }
    // recursive call
    insertWord(child, coord.substr(1));
}
```

```
//for searching word in trie

bool searchWord(TrieNode* root, string word){
    //base case
    if(word.length()==0){
        return root->isTerminal;
    }
    //1 case
    char ch= word[0];
    TrieNode* child= NULL;

    if(root->children[ch]){
        //present
        child = root->children[ch];
    }
    else{
        //absent
        return false;
    }
 //recursive call
bool recursionAns= searchWord(child, word.substr(1));
return recursionAns;
}
```

```
//for deleting a word from trie -

void deleteWord(TrieNode *root, string word){
    //base case
    if(word.length()==0){
    root->isTerminal = false;
    return;
    }

    //1 case
    char ch= word[0];
    TrieNode *child= NULL;

    if(root->children[ch]){
        //present
        child = root->children[ch];
    }
    else{
        //absent
        return;
    }
    //recursive call
    deleteWord(child, word.substr(1));
}
```